RCA 1800
MICROPROCESSORS

Floating-Point Arithmetic Subroutines for RCA COSMAC Microprocessors

MPM-207 Suggested Price $2.00

Floating-Point Arithmetic Subroutines for RCA COSMAC Microprocessors

RCA Solid State

Brussels · Buenos Aires · Hamburg · Madrid · Mexico City · Milan
Montreal · Paris · Sao Paulo · Somerville NJ · Stockholm
Sunbury-on-Thames · Taipei · Tehran · Tokyo

Information furnished by RCA is believed to be accurate and reliable. However, no responsibility is assumed by RCA for its use; nor for any infringements of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of RCA.

Trademark(s) Registered ®
Marca(s) Registrada(s)

Copyright 1978 by RCA Corporation
(All rights reserved under Pan-American Copyright Convention)

Printed in USA/5-78

Foreword

The RCA CDP1802 COSMAC Microprocessor is a CMOS 8-bit register-oriented central processing unit for use in a wide range of stored-program computer systems and products. These applications often require extended arithmetic calculations with possible interfacing of the system to BCD-oriented peripheral hardware. For such applications, the COSMAC Floating-Point Arithmetic Subroutine package CDP18S827 includes a set of 32-bit arithmetic subroutines designed to be operated on COSMAC CDP1802 Microprocessor Systems.

The subroutines are supplied in source code in Level I assembly language and require approximately 2 kilobytes of memory space after assembly. The floating-point binary number is represented by eight exponent bits and 24 mantissa bits. The most significant bit of each indicates the sign. The range of normalized decimal numbers that can be represented by 32 bits is $0.294 \times 10^{-38} \leqslant |FPN| \leqslant 1.7014 \times 10^{38}$.

The COSMAC Floating-Point Arithmetic Subroutine package is available on a floppy diskette CDP18S827 designed for use on RCA Floppy Disk System CDP18S805, a mass memory storage unit designed to work with the CDP18S005 COSMAC Development System (CDS II). It can also be used with Floppy Disk System CDP18S800 and the CDP18S004 COSMAC Development System.

This Manual describes the 18 subroutines provided on the CDP18S827 Diskette and gives detailed information on their application. Ten are arithmetic subroutines, six are utility subroutines, and two are for format conversion. Appropriate selections from the set of subroutines may be made for the calculations required in a specific application.

Users requiring additional information on the available hardware, firmware, and software support systems should refer to the COSMAC technical literature. Reference should be made to the device technical data sheets and to the following publications.

MPM-201 User Manual for the RCA CDP1802 COSMAC Microprocessor

MPM-202 Timesharing Manual for the RCA CDP1802 COSMAC Microprocessor

MPM-203 Evaluation Kit Manual for the RCA CDP1802 COSMAC Microprocessor

MPM-206 Binary Arithmetic Subroutines for RCA COSMAC Microprocessors

MPM-208 Operator Manual for the RCA COSMAC Development System (CDP18S004)

MPM-211 RCA COSMAC Floppy Disk System CDP18S800 Instruction Manual

MPM-212 Instruction Manual for RCA COSMAC Microterminal (CDP18S021

MPM-216 Operator Manual for the RCA COSMAC Development System II CDP18S005

MPM-217 RCA COSMAC Floppy Disk System II CDP18S805 Instruction Manual

MPM-218 Instruction Manual for the RCA COSMAC Micromonitor CDP18S030

MPM-231 Micromonitor Operating System (MOPS) CDP18S831 Users' Guide

Table of Contents

	Page
Foreword	3
Introduction to Floating-Point Arithmetic	5
Why Floating-Point Arithmetic	5
Data Format and Number Representation	5
Range of a 32-Bit Binary Floating-Point Number	7
Mantissa	7
Exponent	7
Limitations	7
Normalization	7
Subroutine Categories and Execution Time	8
Data Storage Convention	11
Two-Operand Functions	11
One-Operand Functions	12
Format Conversion	12
Binary Floating-Point to Decimal Floating-Point Conversion	12
Decimal Floating-Point to Binary Floating-Point Conversion	12
Register Allocations	13
Detailed Description of Floating-Point Arithmetic Subroutines	15
FADD - Addition	15
FSUB - Subtraction	16
FMPY - Multiplication	17
FDIV - Division	18
FSIN - Sine	19
FCOS - Cosine	19
ATAN - Arctangent	20
EXP - Exponential	20
LN - Natural Logarithm	21
SQRT - Square Root	21
Detailed Description of Format-Conversion Subroutines	23
FCBD - Binary to Decimal	23
FCDB - Decimal to Binary	25
Detailed Description of Utility Subroutines	27
COMP - Negation	27
PHSTR - Registers to Stack	27
POPR - Memory to Registers	28
PHZ - Zero to Stack	28
PHST - Constants to Stack	28
DECMA - Decrease R(MA)	28
Accuracy and Errors	29
System Errors	29
User-Caused Errors	29
Accuracy	30
Performance Data	30
Demonstration Program	31
References	32

Introduction to Floating-Point Arithmetic

Why Floating-Point Arithmetic

Scaling and overflow problems in computers are handled in two common ways: fixed-point format or floating-point format arithmetic. In a fixed-point data-word format a word of length N is organized so that the first bit indicates the sign and the remaining N-1 bits give the magnitude of the number. The radix point is fixed and the binary point of the number N is commonly chosen so that the number is always fractional, or

$$-1 \leqslant N \leqslant 1$$

Numbers outside this range require scaling by the programmer who may have difficulty keeping all numbers within this range during computation. Also, overflow, or the generation of a number with a more significant digit than any digit being added, must not be allowed to occur.

To minimize these problems, a floating-point format is frequently preferred because it provides automatic scaling. Arithmetic operations with floating-point number representation, however, are more complicated than operations with fixed-point numbers, and their execution takes longer time. For a given word length, floating-point format trades off some precision for a much greater range in order of magnitude.

Data Format and Number Representation

A floating-point number X is always interpreted to represent a number in the following form:

$$X = m.r^e$$

where m, the mantissa (also referred to as the coefficient or fractional part) is stored in one register. Another register contains the exponent e (also called characteristic). The radix, or base r, and the radix point in the mantissa are always assumed to be a fixed value dependent upon a particular implementation.

Examples of floating-point representations are:

Decimal System:

$1{,}472 = 0.1472 \times 10^4$
$-31.2 = -0.312 \times 10^2$
$0.0364 = 0.364 \times 10^{-1}$

Binary System:

$1101 = 0.1101 \times 2^4$
$-1.1101 = -0.11101 \times 2^1$
$0.0110 = 0.1100 \times 2^{-1}$

In the software package described in this Manual, RCA COSMAC Floating - Point Arithmetic Subroutine Diskette CDP18S827, the floating-point numbers are represented by 32 bits: 24 bits for the mantissa and 8 bits for the exponent in base two. Both the mantissa and the exponent are in 2's-complement notation. The data format is depicted in Fig. 1.

Example 1

Consider the fixed-point hexadecimal number

F8A0.1345

The floating-point representation is

$0.F8A01345 \times 2^{10}{}_H$

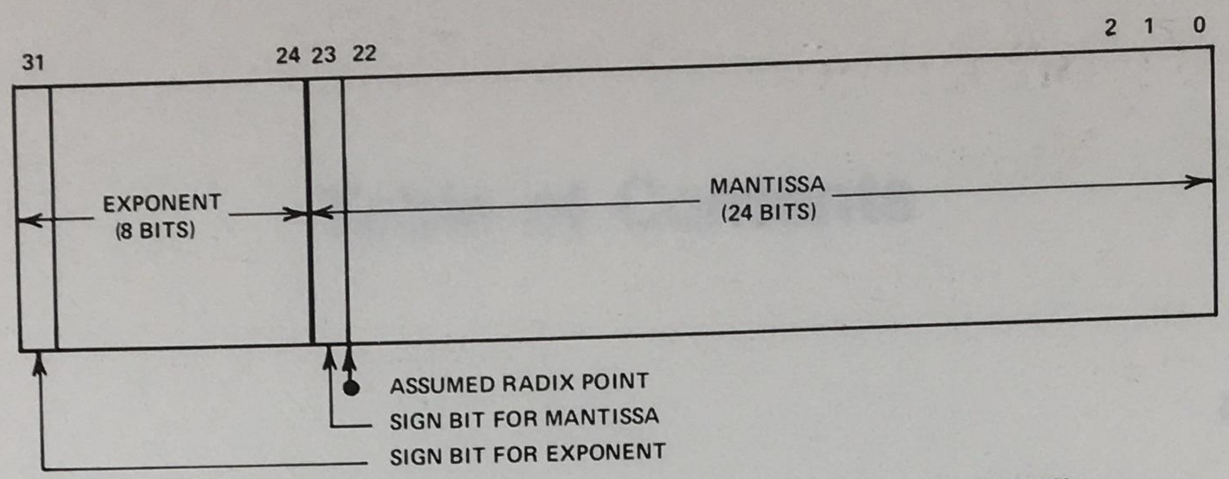


Fig. 1 – Data format for floating-point numbers.

In binary, the fixed-point expression would be written

1111 1000 1010 0000. 0001 0011 0100 0101

and the floating-point expression becomes

$0.111\ 1100\ 0101\ 0000\ 0000\ 1001\ 1010\ 0010\ 1 \times 2^{010000}$

The 23 most significant digits of the number would be stored in the format shown in Fig. 2 in binary, or in

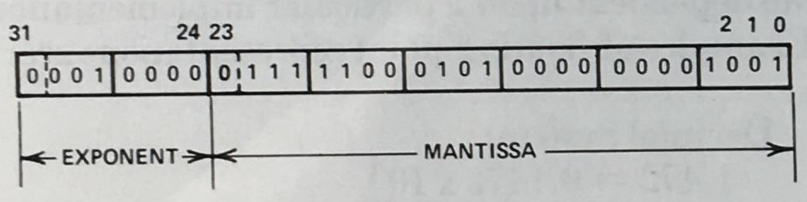


Fig. 2 – Floating-point notation storage format in binary of fixed-point hexadecimal number F8A0.1345.

hexadecimal as shown in Fig. 3. Note the truncation of the least significant bits that do not fit the 24-bit mantissa.

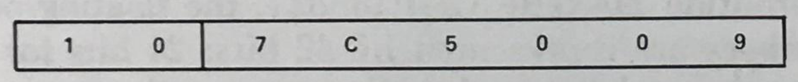


Fig. 3 – Floating-point notation storage format in hexadecimal of fixed-point hexadecimal number F8A0.1345.

Example 2

Fixed-point number in hexadecimal notation:

-0.0000054F21A2

Floating-point representation is:

$-0.54F21A2 \times 2^{-14}{}_H$

In binary, the fixed-point number would be written:

-0.0000 0000 0000 0000 0000 0101 0100 1111
0010 0001 1010 0010

and the floating-point number in 2's-complement would be

$1.1010\ 1011\ 0000\ 1101\ 1110\ 0101\ 1110 \times 2^{11101100}$

As explained later, the above number is not in normalized form. After normalization, the value becomes

$1.010\ 1011\ 0000\ 1101\ 1110\ 0101\ 1110 \times 2^{11101011}$

The number would be stored in the format shown in Fig. 4.

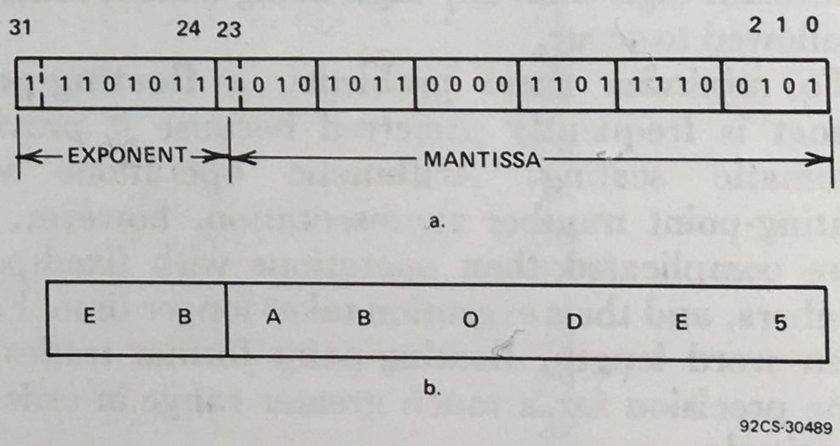


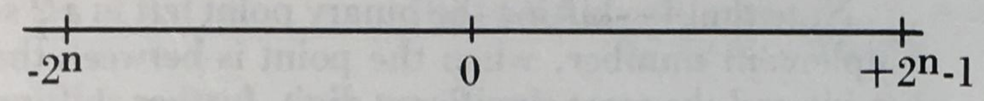
Fig. 4 – a. Binary storage format of –0.0000054F21A2.
b. Hexadecimal storage format of same number.

Range of a 32-Bit Binary Floating-Point Number

As already stated, one of the advantages in using floating-point arithmetic is that a wide range of numbers can be represented.

Mantissa

The range of an n-bit integer in 2's-complement notation is



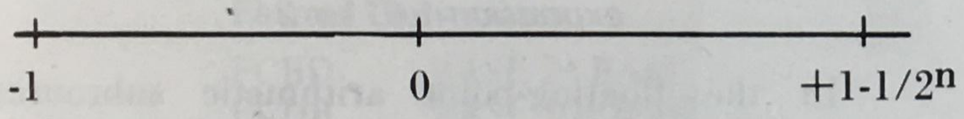
If n = 7, the most positive number is

$$01111111_2 = 127_D$$

and the most negative number is

$$10000000_2 = 128_D$$

To convert this number line so that the mantissa is in the form $-1 <$ mantissa < 1, divide by 2^n. Thus, the number becomes



For a 24-bit mantissa, including the sign bit, n = 23,

Hence:

$-1 \leqslant \text{mantissa} \leqslant 1 - 1/2^{23}$ equivalently,

$-1 \leqslant \text{mantissa} \leqslant (1 - 2^{-23})$

Exponent

Similarly, the range of an exponent in 2's-complement notation is

$$-2^n \leqslant \text{exponent} \leqslant 2^n - 1$$

For an 8-bit exponent, including sign bit, n = 7.

Hence:

$$-128_D \leqslant \text{exponent} \leqslant 127_D$$

Therefore, the range of a 32-bit binary floating-point number (FPN) with 24-bit mantissa and 8-bit exponent is

$$-1 \times 2^{127} \leqslant \text{FPN} \leqslant (1 - 2^{-23}) \times 2^{127}$$

and is depicted in Fig. 5.

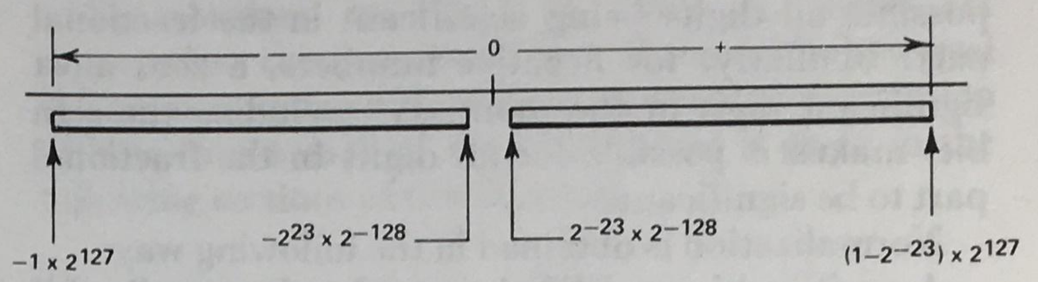


Fig. 5 – Number range of 32-bit floating-point number with 24-bit mantissa and 8-bit exponent.

In decimal, the range is approximately

$$-1.70141 \times 10^{38} \leqslant \text{FPN} \leqslant 1.70141 \times 10^{38}$$

As illustrated in Fig. 5, the absolute range of a 32-bit FPN is

$$(2^{-23}) \times 2^{-128} \leqslant |\text{FPN}| \leqslant (1 - 2^{-23}) \times 2^{127}$$

which in decimal representation is

$$3.503 \times 10^{-46} \leqslant |\text{FPN}| \leqslant 1.70141 \times 10^{38}$$

Limitations

The user should be aware that, by the nature of binary floating-point representation, the floating-point numbers with a given number of bits do not form a continuous set. Therefore, not all the numbers within the previously defined range can be represented exactly by the 32 bits.

This type of error is called **quantizing error** and is discussed in more detail later. The errors are not distributed uniformly on the number scale. The magnitude of the quantizing error varies with the number represented and is related to the magnitude of that number.

Normalization

There are infinite number of ways to represent a floating-point number because the decimal point may be shifted any number of positions in either direction.

Highest precision for a given number of bits in the word format is achieved if a maximum number of significant digits is stored in the register (achieved by adjusting the exponent and mantissa to maintain

maximum significance). This scaling of a floating-point number is called **normalizing.**

A non-zero most significant digit in the mantissa (excluding the sign bit) for a positive number, makes possible all digits being significant in the fractional part. Similarly, for negative numbers, a zero most significant digit in the mantissa (excluding the sign bit) makes it possible for all digits in the fractional part to be significant.

Normalization is obtained in the following way:

A positive binary FPN is considered normalized if there is no more than one leading "0" in the mantissa bits of the number.

A negative binary FPN is considered normalized if there is no more than one leading "1" in its mantissa bits. **Thus, a binary FPN is said to be normalized if the most significant two bits of the mantissa are different** (bits 23 and 22 in Fig. 1).

To normalize an unnormalized binary floating-point number, the mantissa bits are shifted left until the two left-most bits are different. The exponent is adjusted accordingly by having the number of shifts subtracted from it.

Examples of scaling are given below:

Example No. 1:

$4_{10} = 0100_2$

The same positive binary number can be expressed in various ways

0100000.	x	2^{-3}	=	0100000.	x	2^{101}
01000.	x	2^{-1}	=	01000.	x	2^{111}
0.100	x	2^{3}	=	0.100	x	2^{011} *
0.0100	x	2^{4}	=	0.0100	x	2^{0100}
0.00100	x	2^{5}	=	0.00100	x	2^{0101}

* (only normalized form)

The normalized expression 0.100×2^3 in an 8-bit data format would be stored as shown in Fig. 6.

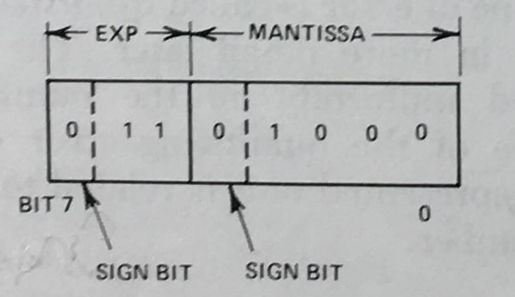


Fig. 6 – Storage format for normalized expression 0.100×2^3.

Example No. 2

$-5_{10} = -0101_2$

In 2's-complement notation

$-5_{10} = 1011_2$

The negative number can be expressed variously as shown below:

10110.	x	2^{-1}	=	10110.	x	2^{111}
1011000.	x	2^{-3}	=	1011000.	x	2^{101}
10.11	x	2^{2}	=	10.11	x	2^{010}
1.011	x	2^{3}	=	1.0110	x	2^{011} *
1.1011	x	2^{4}	=	1.1011	x	2^{0100}
1.11011	x	2^{5}	=	1.11011	x	2^{0101}
1.111011	x	2^{6}	=	1.111011	x	2^{0110}

* (only normalized form)

Note that in shifting the binary point left in a 2's-complement number, when the point is between the sign bit and the most significant digit, further shifting to the left requires the insertion of ones to the left of the most significant data bit. The normalized expression 1.011×2^3 in an 8-bit data format would be stored as shown in Fig. 7.

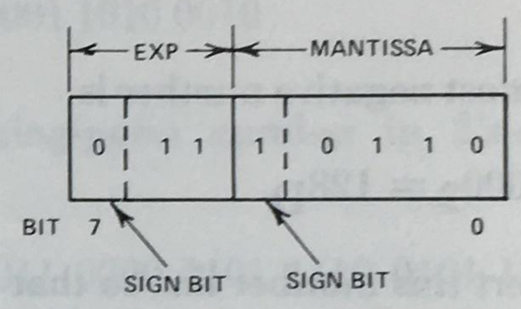


Fig. 7 – Storage format for normalized expression 1.011×2^3.

In the floating-point arithmetic subroutine package CDP18S827, the results derived from all the mathematical functions as well as the format conversions are stored in a normalized form. Although the arguments of these functions do not have to be normalized prior to the execution, the user should be aware that failure to normalize may result in loss of accuracy.

Subroutine Categories and Execution Time

The binary floating-point categories implemented in the CDP18S827 package can in general be divided into three groups: arithmetic functions, format-conversion functions, and utility functions.

The arithmetic functions include the four basic ones: add, subtract, multiply, and divide, as well as six trancendental functions -- sine, cosine, arc-tangent, exponential, natural logarithm, and square root.

The format-conversion functions can be used to convert binary floating-point numbers to and from decimal floating-point numbers.

The utility functions are provided for the convenience of the user. They can save a group of operand registers or a four-byte floating-point constant onto a stack. A memory address pointer is set to the floating-point number if a constant is saved. Conversely, the saved floating-point number can be popped off from the stack and restored into operand registers.

A floating-point number stored in the operand registers can also be negated by using a negation utility routine.

A list of the functions included in the 32-bit floating-point package is given in Table I and the utility routines are listed in Table II. Table III summarizes the execution times of the arithmetic and format-conversion subroutines. A detailed description of how to use each of the subroutines is given in the following sections of this Manual.

TABLE I – Arithmetic and Format-Conversion Functions Included in 32-Bit Floating-Point Package CDP18S827

Arithmetic Functions

NAME	FUNCTION	OPERANDS	DOMAIN X (operands)	RANGE Y (result)
FADD	ADDITION	2	FULL	FULL
FSUB	SUBTRACTION	2	FULL	FULL
FMPY	MULTIPLICATION	2	FULL	FULL
FDIV	DIVISION	2	FULL	FULL
FSIN	SINE	1	FULL	$-1 \leqslant Y \leqslant 1$
FCOS	COSINE	1	FULL	$-1 \leqslant Y \leqslant 1$
ATAN	ARCTANGENT	1	$X \geqslant 0$	$0 \leqslant Y \leqslant 90$
EXP	EXPONENTIAL	1	$-88.72 < X < 88.03$	FULL
LN	NATURAL LOG	1	$X > 0$	$-88.72 \leqslant Y \leqslant 88.03$
SQRT	SQUARE ROOT	1	$X \geqslant 0$	$0 \leqslant Y$

Format Conversions

FCBD	BASE 2→ BASE 10	1	FULL	*
FCDB	BASE 10→ BASE 2	1	*	FULL

*Entire decimal range of a 32-bit binary floating-point number.

FULL implies X or Y such that $(2^{-23}) \times 2^{-128} \leqslant |\,FPN\,| \leqslant (1-2^{-23}) \times 2^{127}$

Internal Subroutines

ADD	DICE
CORDIC	L90
DSHL	SUB

TABLE II – Utility Subroutines

NAME	FUNCTION	OPERANDS
COMP	NEGATION	1
PHSTR	REGISTERS → STACK	1
PHST	CONSTANTS → STACK	1
POPR	STACK → REGISTERS	1
PHZ	ZERO → STACK	1
DECMA	R(MA) – #XX	1

TABLE III – Summary of Execution Times of Floating-Point Arithmetic and Format-Conversion Subroutines

SUBROUTINE	2-MHz CLOCK		6.4-MHz CLOCK	
	BEST (ms)	WORST (ms)	BEST (ms)	WORST (ms)
FADD	1.7	25	0.53	7.8
FSUB	1.3	26	0.41	8.1
FMPY	140	152	43.8	47.5
FDIV	96	104	30	32.5
FSIN *	360	370	113	116
FCOS *	325	360	102	113
ATAN *	275	350	85.9	109
EXP	230	400	71.9	125
LN	250	600	78.1	188
SQRT	495	1000	155	312
FCBD	7.5	150	2.3	46.9
FCDB	24	5000	7.5	1600

* For $0^o < \theta < 360^o$

These times were determined by taking an ad hoc sample of large and small numbers and performing an operation upon them. Absolute best and worst case values may vary from those listed here and elsewhere in this Manual.

Data Storage Convention

There are two basic types of operations for arithmetic functions. The first one is the register-to-memory operation where two operands are required. The second type is register operation where only a single operand is needed. After a computation, the result is stored back into the operand register for either type of operation. For format conversion, the signs and digits of the decimal floating-point number is stored in a user-defined RAM area. On the other hand, a binary floating-point number is stored into two 16-bit registers.

The following section gives a more detailed explanation of how the operands should be stored for each type of operation.

Two-Operand Functions

For addition, subtraction, multiplication, and division, the operands prior to the calling of the routines and the results after the execution of the functions are stored as illustrated in Fig. 8.

The 32-bit operand #1 is stored in two 16-bit registers that are previously defined by the user. The 8-bit exponent of the operand #1 occupies the most significant 8-bits of the two operand registers. The remaining 8 bits of the register store the most significant 8 bits of the 24-bit mantissa. The least significant 16 bits of the mantissa are stored in the other register.

Operand #2 is stored in memory as four consecutive memory bytes. A 16-bit operand address register should be pointing to the exponent byte of operand #2. The three mantissa bytes are stored in the memory locations following the exponent, with the most significant bits stored ahead of the less significant.

After computation, the result is stored back into the same two registers used for operand #1 with the same exponent-mantissa sequence.

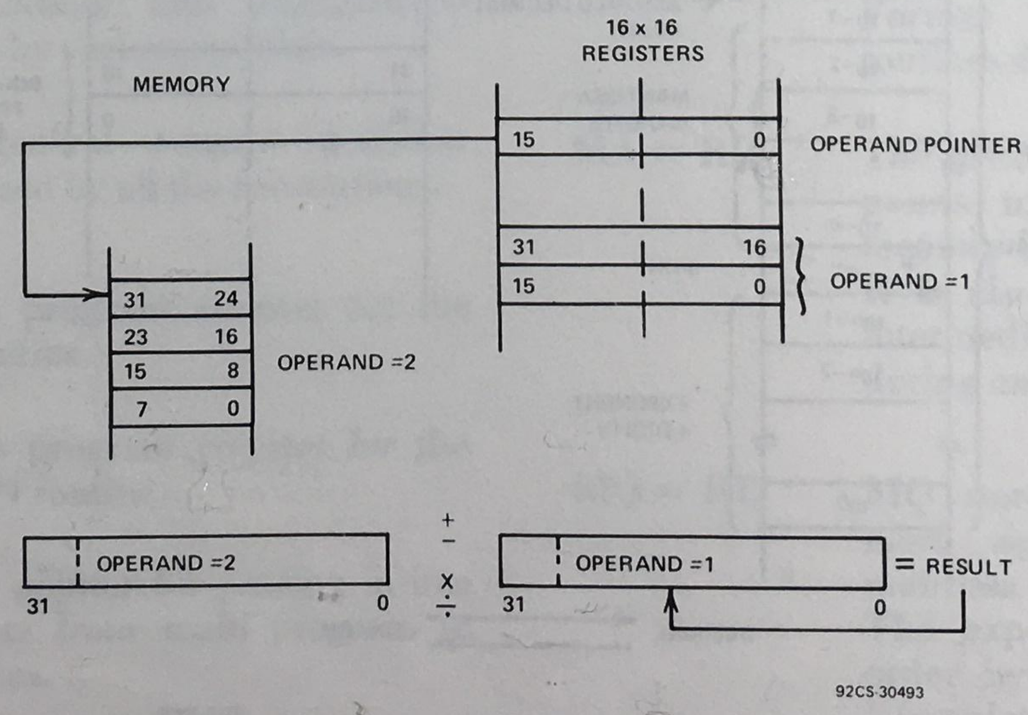


Fig. 8 — Operation of two-operand function.

One-Operand Functions

One-operand functions include sine, cosine, arctangent, exponential, natural logarithm, and square root. To perform any one of these functions, the user should store the single operand in the same manner as operand #1 in the case above. After computation, the result is stored back into the same set of registers. This operation is illustrated in Fig. 9.

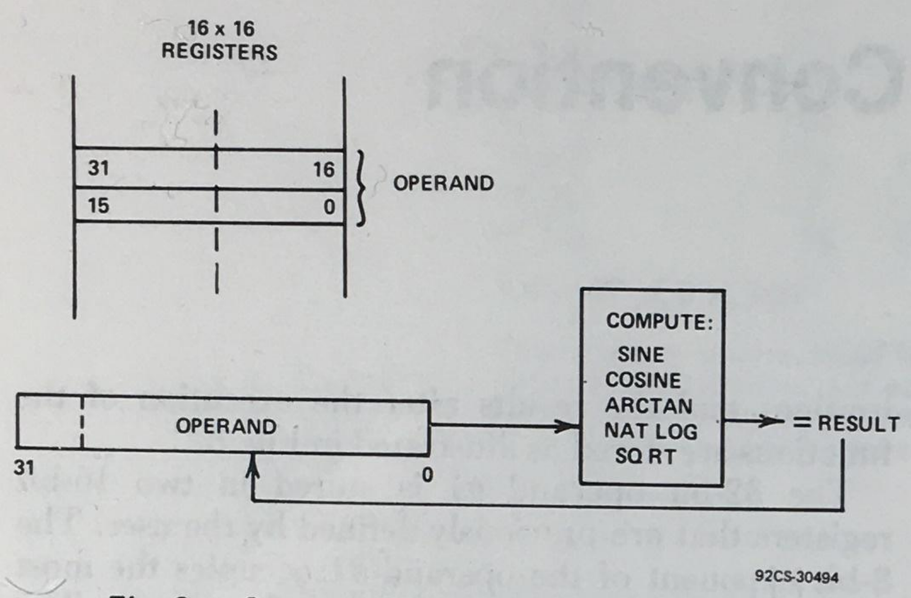


Fig. 9 — Operation of one-operand function.

Format Conversions

Two routines are provided in the floating-point arithmetic package to convert binary FPN's to and from decimal FPN's, typically for digital display and/or terminal type I/O.

Binary Floating-Point to Decimal Floating-Point Conversion

Two-operand registers contain the 32-bit binary floating-point number that is to be converted. The user passes the memory address as an in-line argument indicating where the signs and the digits of the converted decimal floating-point number will be stored.

The user also provides the conversion routine with the desired length m of the resultant mantissa digits, as well as the desired length n of the exponent digits.

After the conversion, the decimal floating-point number is stored in memory with one byte per digit for m + n + 2 digits. The two extra bytes are for the sign of the mantissa and the sign of the exponent. A decimal point is assumed to the right of the sign byte of the decimal floating-point number.

Decimal Floating-Point to Binary Floating-Point Conversion

The decimal to binary conversion works in a similar fashion as described above, except that the data flow is reversed. These format conversions are depicted in Fig. 10.

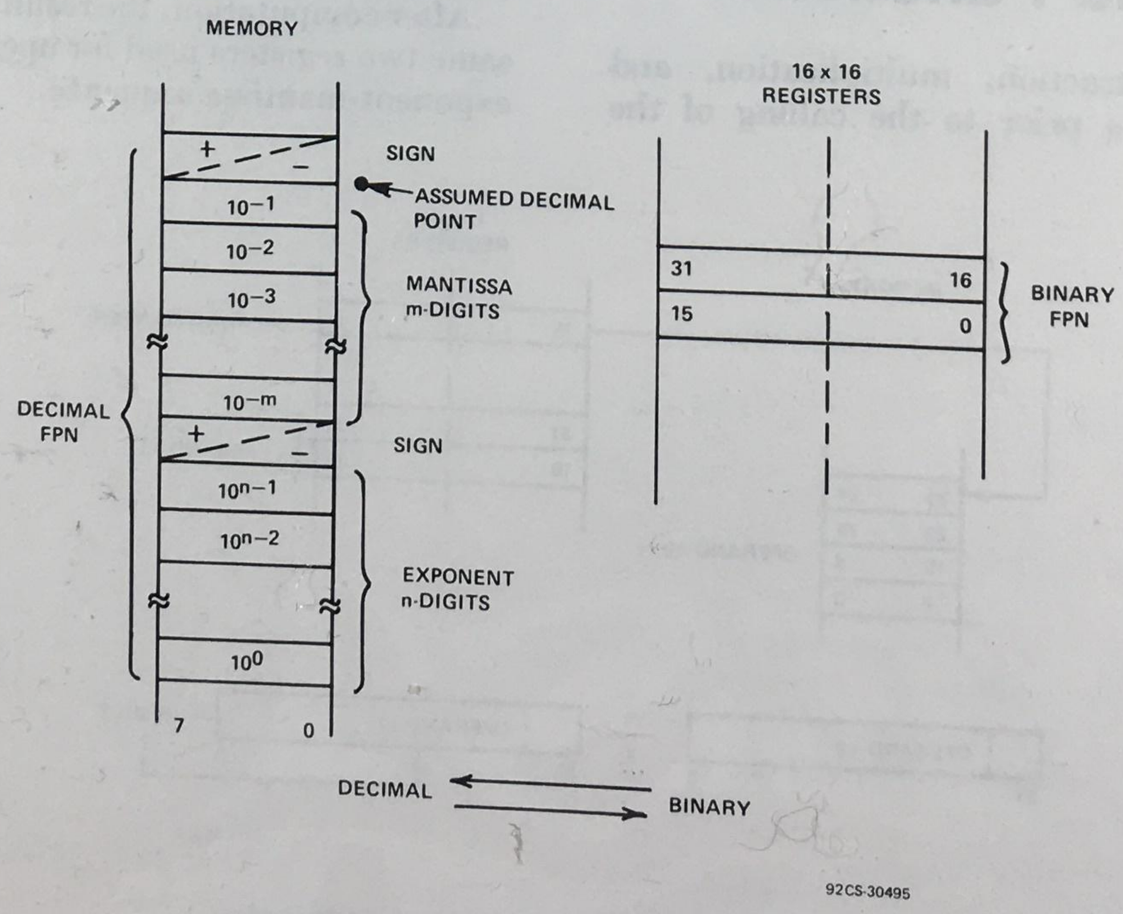


Fig. 10 — Operation of format-conversion function.

Register Allocations

Six registers are used by all the subroutines for storing operands, addresses, and intermediate data. Five other registers are used as standard program counter, stack pointer, CALL and RETURN subroutine program counters, and parameter passing pointers. The use of the Standard Call and Return Technique (SCRT) described in the User Manual for the CDP1802 COSMAC Microprocessor, MPM-201, is assumed.

Symbolic names of these registers are used throughout the arithmetic package. Thus, it is feasible to alter the actual register allocation by assigning the set of registers to be used at the user's discretion.

The standard register assignment for all the subroutines are:

SP = R2 — SP is used as a stack pointer to a free RAM area. This RAM area is used as temporary storage for subroutine return address and intermediate data used by various routines.

PC = R3 — PC is general purpose program counter used by all the subroutines.

CALL = R4 — CALL is program counter for the CALL routine.

RETN = R5 — RETN is program counter for the RETURN routine.

LINK = R6 — LINK is pointer for passing in-line parameters from main program to subroutines.

SR = RA — The SCRATCH register is used for temporary storage in various arithmetic and conversion functions. The utility routines do not alter this register.

CN = RB — The CONTROL register and COUNTER is used by various routines to pass a control word to an internal routine for computation of various functions. It is also used as iteration counter during computations. The register is used by all subroutines except the add, subtract, and utility routines.

TP = RC — A TABLE POINTER register is used for pointing to a table of constants that are used during computations of transcendental functions.

The register is used by all arithmetic functions except add, subtract, multiply, and divide routines. Format conversions and utility routines do not alter this register.

MA = RD — The MEMORY ADDRESS register points to an operand stored in memory for two-operand operations. It is also used for pointing to an intermediate floating-point number during one-operand operations.

MQ = RE — MQ stores the exponent and the most significant 8 bits of the mantissa of an operand or result. The exponent occupies the high-order byte; the mantissa uses the low-order byte of the register. MQ with AC store a full 32-bit floating-point number. MQ is used by all the routines.

AC = RF	AC is used to store the least significant 16 bits of the mantissa of a 32-bit floating-point operand or result. It is used by all the subroutines.

Note: The notation (R(MQ)) ; R(AC)) is used to denote a pair of registers that together hold a 32-bit floating-point number.

Operations that involve placing numbers onto the stack, though transparent to the user, manipulate R(SP). Care, therefore, should be exercised when interrupts are to be used. It is recommended that interrupts be disabled during the execution of any floating-point subroutines and then re-enabled, if desired.

Detailed Description of Floating-Point Arithmetic Subroutines

FADD - Addition

FADD is a subroutine that adds two binary floating point numbers. One of the floating point numbers should be stored in registers (MQ ; AC). Register R(MQ) contains the exponent and the most significant 8-bits of the mantissa; register R(AC) contains the least significant 16 bits of the mantissa. The other floating-point number is stored in memory. The 32-bit number is stored as four consecutive bytes in memory with R(MA) pointing to the exponent byte followed by three bytes of mantissa. The result is stored back in registers (R(MQ) ; R(AC)) in a normalized form.

1. **To Call FADD:**

```
SEP CALL
,A(FADD)   ..R(MA) has been previously
           ..loaded with the address of
           ..the operand in memory.
```

2. **Registers Used by FADD:**

R(AC), R(MQ), R(MA), R(SR)

3. **Error Indication:**

Upon return from subroutine FADD, the flag DF is set if the sum is overflowed or underflowed. Otherwise DF is reset to zero.

4. **Number Range:**

Domain X:

$800000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$
$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y:

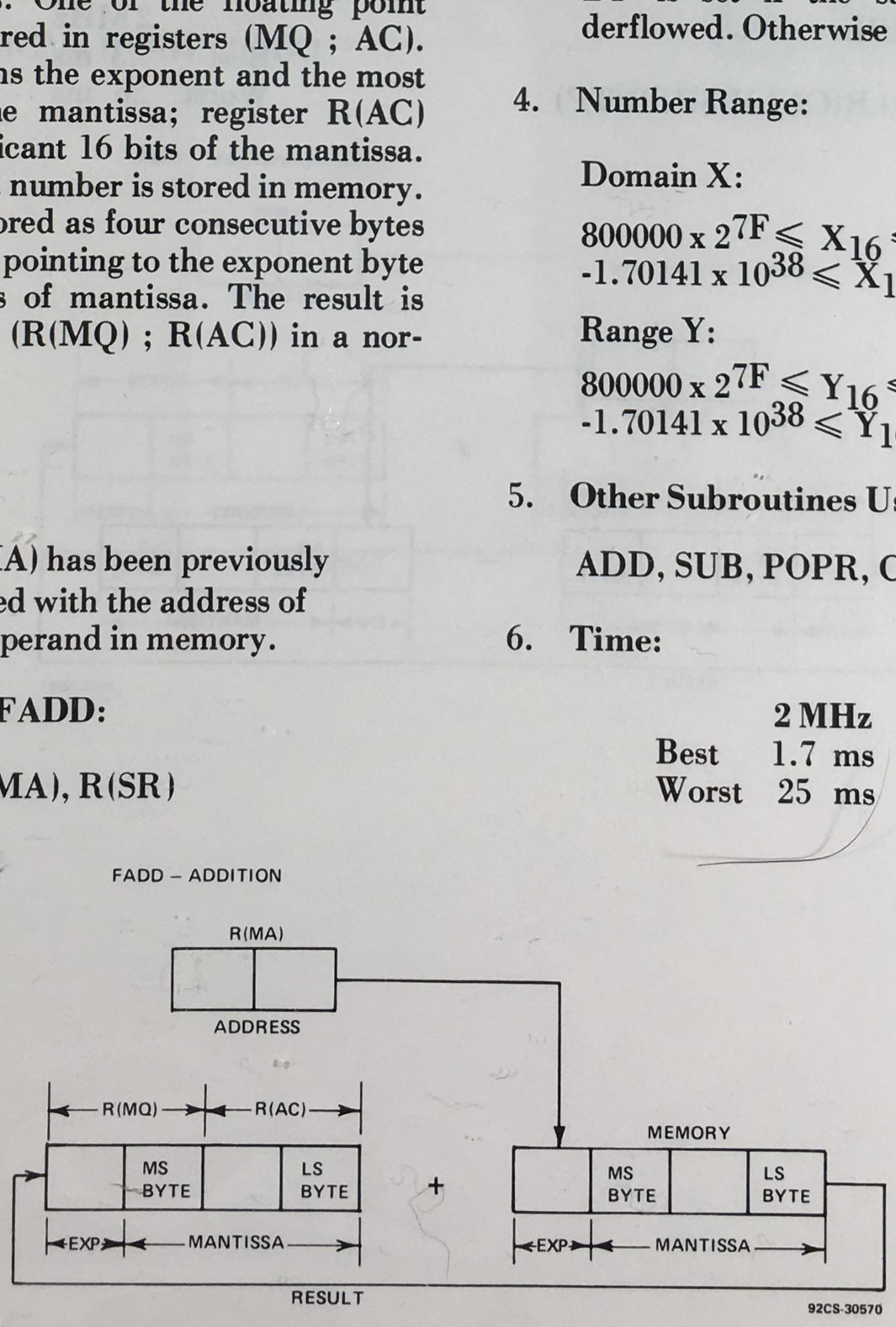
$800000 \times 2^{7F} \leqslant Y_{16} \leqslant 7FFFFF \times 2^{7F}$
$-1.70141 \times 10^{38} \leqslant Y_{10} \leqslant 1.70141 \times 10^{38}$

5. **Other Subroutines Used:**

ADD, SUB, POPR, COMP

6. **Time:**

	2 MHz	6.4 MHz
Best	1.7 ms	0.53 ms
Worst	25 ms	7.8 ms



FSUB - Subtraction

FSUB is a subroutine that subtracts a binary floating-point number in memory, pointed to by an address register R(MA), from a floating-point number in registers (R(MQ) ; R(AC)). The register R(MQ) contains the exponent and the most significant 8 bits of the mantissa of the minuend. Register R(AC) contains the least significant 16 bits of the mantissa of the minuend. Register R(MA) points to the exponent byte of the subtrahend stored in memory. The three mantissa bytes of the subtrahend in memory are stored adjacent to the exponent byte. The result is stored back into registers (R(MQ) ; R(AC)) in a normalized form.

1. **To Call FSUB:**

```
SEP CALL
,A(FSUB)   ..R(MA) contains the
           ..address of the operand
           ..in memory.
```

2. **Registers Used by FSUB:**

R(AC), R(MQ), R(MA),R(CN),R(SR),R(TP)

3. **Error Indication:**

An error flag is set if the result has overflowed or underflowed. Otherwise the flag is reset to zero.

4. **Number Ranges:**

Domain X:

$800000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y:

$800000 \times 2^{7F} \leqslant Y_{16} \leqslant 7FFFFF \times 2^{7F}$

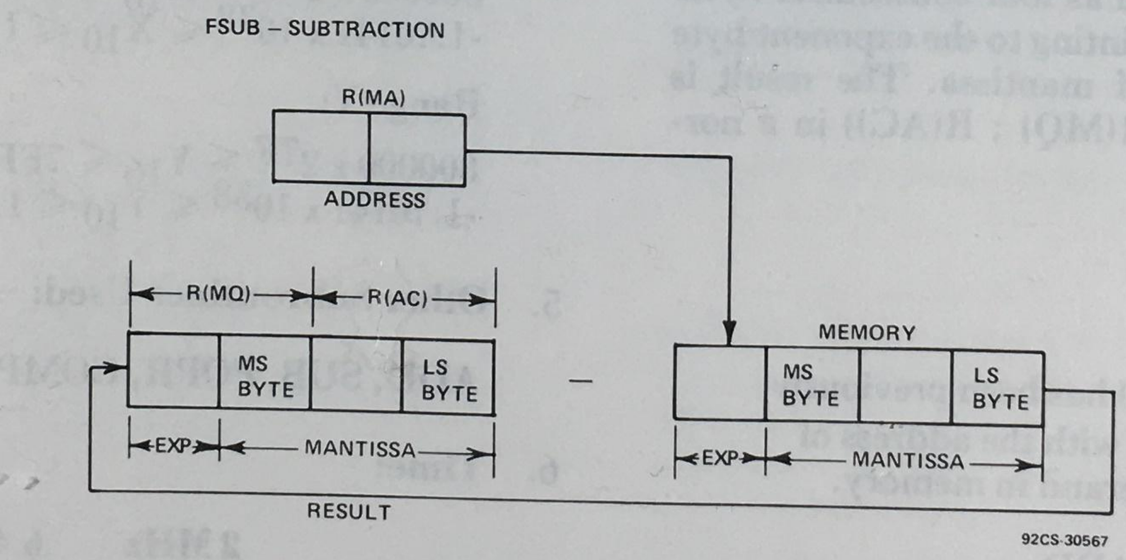
$-1.70141 \times 10^{38} \leqslant Y_{10} \leqslant 1.70141 \times 10^{38}$

5. **Other Subroutines Used:**

ADD, SUB, POPR, COMP

6. **Time:**

	2 MHz	6.4 MHz
Best	1.3 ms	0.41 ms
Worst	26 ms	8.1 ms



FMPY - Multiplication

FMPY is a subroutine that multiplies a binary floating-point number in registers (R(MQ) ; R(AC)) by a binary floating-point number in memory, pointed to by the address register R(MA). Register R(MQ) contains the exponent and the most significant 8 bits of the mantissa of the multiplicand. Register R(AC) contains the least significant 16 bits of the mantissa. The multiplier is stored in memory as four consecutive bytes. R(MA), the address register, points to the exponent byte of the multiplier. The 32-bit product is stored back into registers (R(MQ) ; R(AC)) in a normalized form.

1. **To Call FMPY:**

```
SEP CALL
,A(FMPY)   ..R(MA) should contain
           ..the address of the
           ..multiplier.
```

2. **Registers Used by FMPY:**

R(AC), R(MQ), R(CN), R(SR), R(MA)

3. **Error Indication:**

The flag DF is set if the product has overflowed or underflowed. Otherwise DF is reset to zero.

4. **Number Range:**

Domain X:

$80000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

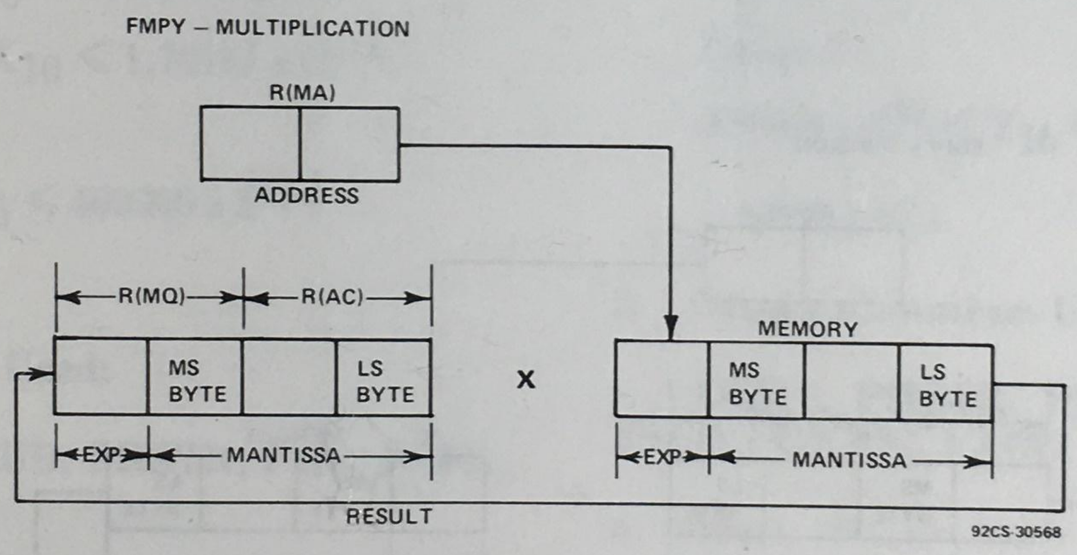
Range Y: Same as domain

5. **Other Subroutines Used:**

PHSTR, POPR, COMP, PHZ, CORDIC, DECMA

6. **Time:**

	2 MHz	6.4 MHz
Best	140 ms	43.8 ms
Worst	152 ms	47.5 ms



FDIV - Division

FDIV is a subroutine that divides a binary floating-point number in registers (R(MQ) ; R(AC)) by a binary floating-point number in memory, pointed to by an address register R(MA). The exponent and the most significant 8 bits of the mantissa of the dividend are stored in register R(MQ). The least significant 16 mantissa bits are stored in R(AC). The divisor is stored in memory as four consecutive bytes. The result is stored back into registers (R(MQ)) ; R(AC)) after the division. In contrast to fixed-point division, in which a quotient and a remainder are usually obtained, only the quotient part is generated in floating-point division.

1. **To Call FDIV:**

```
SEP CALL
,A(FDIV)   ..R(MA) contains the address
           ..of the divisor before the
           ..division.
```

2. **Registers Used by FDIV:**

R(AC), R(MQ), R(MA), R(SR), R(CN)

3. Error Indication:

The flag DF is set if the quotient is overflowed or underflowed. Otherwise DF is reset to zero.

4. **Number Range:**

Domain X:

$800000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y: Same as domain

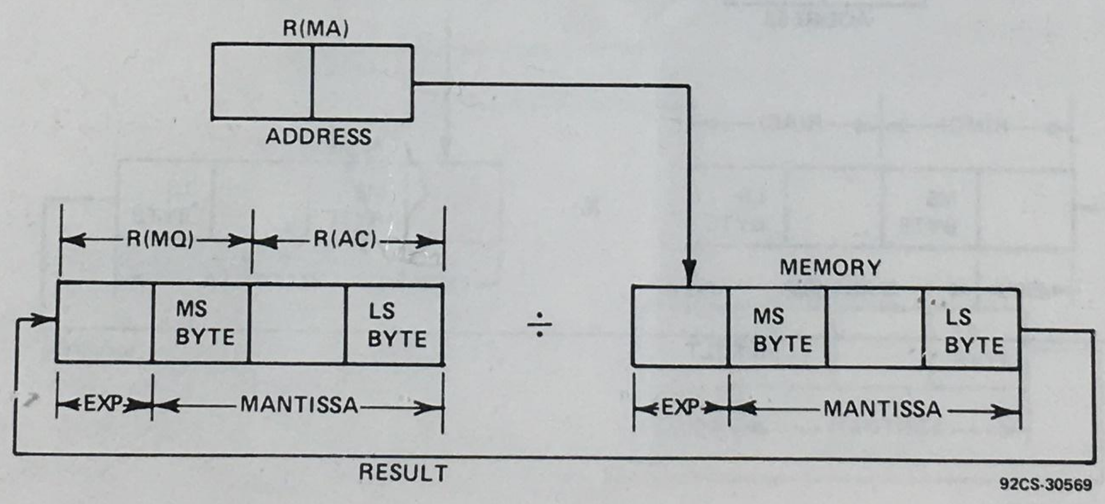
5. **Other Subroutines Used:**

PHSTR, POPR, COMP, CORDIC

6. **Time:**

	2 MHz	6.4 MHz
Best	96 ms	30 ms
Worst	104 ms	32.5 ms

FDIV – DIVISION



FSIN - Sine

FSIN is a subroutine that computes the sine of a 32-bit binary floating-point number in registers (R(MQ) ; R(AC)). The units for this number are assumed to be degrees. R(MQ) contains the exponent and the most significant 8 bits of the mantissa; R(AC) contains the least significant 16 bits of the mantissa. After the computation, the result is stored back into the same set of registers (R(MQ) ; R(AC)) in a normalized form.

1. **To Call FSIN:**

```
SEP CALL
,A(FSIN)
```

2. **Registers Used by FSIN:**

R(AC), R(MQ), R(CN), R(SR), R(MA), R(TP)

3. **Error Indication: None**

4. **Number Range:**

Domain X:

$800000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y:

$800000 \times 2^{00} \leqslant Y_{16} \leqslant 400000 \times 2^{01}$

$-1 \leqslant Y_{10} \leqslant 1$

5. **Other Subroutines Used:**

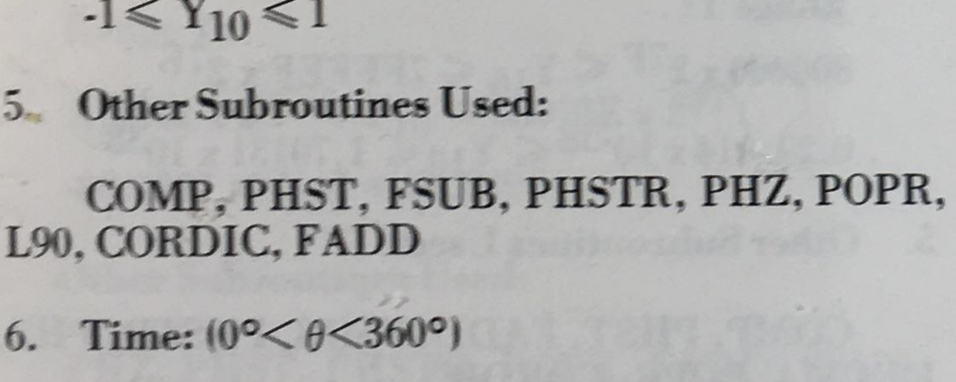
COMP, PHST, FSUB, PHSTR, PHZ, POPR, L90, CORDIC, FADD

6. **Time:** $(0° < \theta < 360°)$

	2 MHz	6.4 MHz
Best	360 ms	113 ms
Worst	370 ms	116 ms

Note: Although FSIN will compute an argument within the domain defined above, the argument should be scaled down to less than or equal to 90 degrees before the actual computation. Prescaling within the subroutine can be time consuming.

FSIN – SINE



FCOS - Cosine

FCOS is a subroutine that computes the cosine of a binary floating-point number stored in registers (R(MQ) ; R(AC)). This floating-point number is assumed to be in degrees. After the computation, the resultant value is stored back into registers (R(MQ) ; R(AC)) in a normalized form.

1. **To Call FCOS:**

```
SEP CALL
,A(FCOS)
```

2. **Registers Used by FCOS:**

R(AC), R(MQ), R(MA), R(SR), R(CN), R(TP)

3. **Error Indication: None**

4. **Number Range:**

Domain X:

$800000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y:

$800000 \times 2^{00} \leqslant Y_{16} \leqslant 400000 \times 2^{01}$

$-1 \leqslant Y_{10} \leqslant 1$

5. **Other Subroutines Used:**

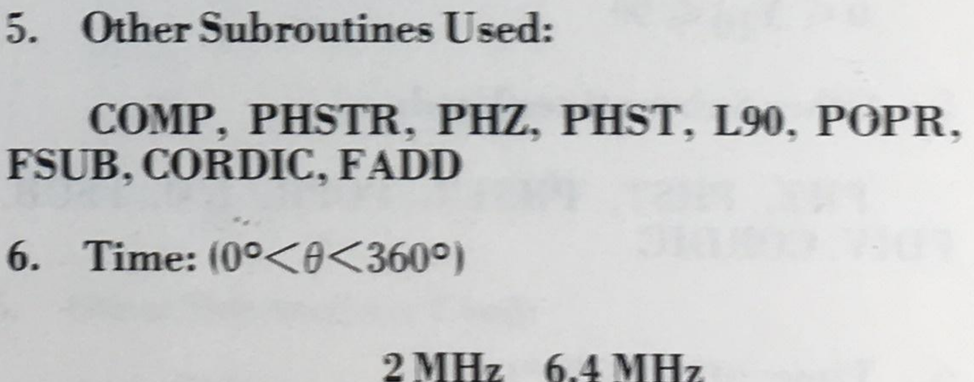
COMP, PHSTR, PHZ, PHST, L90, POPR, FSUB, CORDIC, FADD

6. **Time:** $(0° < \theta < 360°)$

	2 MHz	6.4 MHz
Best	325 ms	102 ms
Worst	360 ms	113 ms

Note: If the argument is much larger than 360 degrees the prescaling process could be time consuming. Refer to FSIN section.

FCOS – COSINE



ATAN - Arctangent

ATAN is a subroutine that computes the arctangent of a binary floating-point number stored in register (R(MQ)) ; R(AC)). R(MQ) stores the exponent and the most significant 8 bits of the mantissa. R(AC) stores the least significant 16 bits of the mantissa. After the computation, the result is stored in registers (R(MQ)); R(AC)) in a normalized form. The result is in degrees.

1. **To Call ATAN:**

```
SEP CALL
,A(ATAN)
```

2. **Registers Used by ATAN:**

R(AC), R(MQ), R(MA), R(CN), R(SR), R(TP)

3. **Error Indication:** None

4. **Number Range:**

Domain X:

$0 \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$0 \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y:

$0 \leqslant Y_{16} \leqslant 5A0000 \times 2^{07}$

$0 \leqslant Y_{10} \leqslant 90$

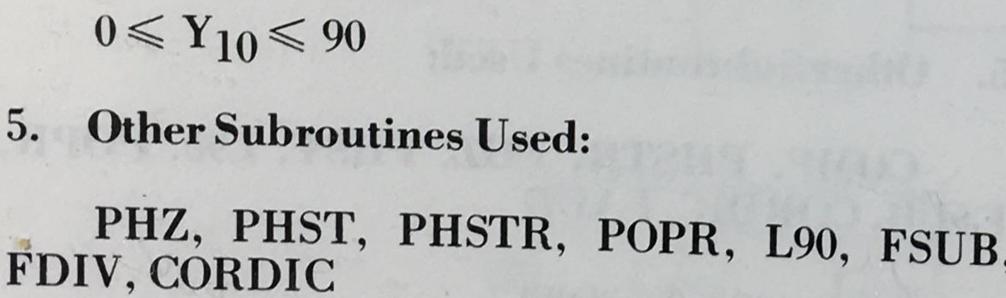
5. **Other Subroutines Used:**

PHZ, PHST, PHSTR, POPR, L90, FSUB, FDIV, CORDIC

6. **Time:** ($0° < \theta < 90°$)

	2 MHz	6.4 MHz
Best	275 ms	85.9 ms
Worst	350 ms	109 ms

ATAN – ARCTANGENT



EXP - Exponential

EXP is a subroutine that computes the function e to the power of a 32-bit floating-point number in registers (R(MQ) ; R(AC)). The exponent and the most significant 8 bits of the mantissa are stored in R(MQ). Register R(AC) stores the least significant 16 bits of the mantissa. After the computation, the result is stored back into registers (R(MQ) ; R(AC)) in normalized form.

1. **To Call EXP:**

```
SEP CALL
,A(EXP)
```

2. **Registers Used by EXP:**

R(AC), R(MQ), R(MA), R(CN), R(SR), R(TP)

3. **Error Indication:**

Upon return from EXP, the DF flag is set if the result is overflowed or underflowed. Otherwise DF is reset to zero.

4. **Number Range:**

Domain X:

$A74701 \times 2^{07} \leqslant X_{16} \leqslant 58078E \times 2^{07}$

$-88.7227 \leqslant X_{10} \leqslant 88.0294$

Range Y:

$800000 \times 2^{7F} \leqslant Y_{16} \leqslant 7FFFFF \times 2^{7F}$

$0.293914 \times 10^{-38} \leqslant Y_{10} \leqslant 1.70131 \times 10^{38}$

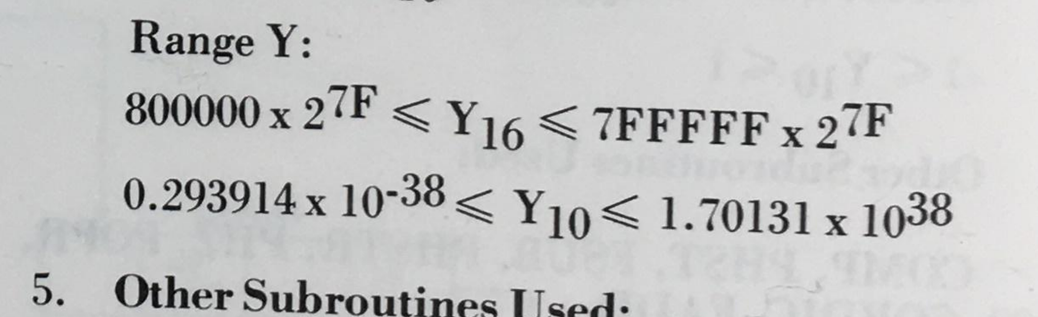
5. **Other Subroutines Used:**

COMP, PHST, FADD, FSUB, PHSTR, PHZ, DECMA, POPR, CORDIC

6. **Time:**

	2 MHz	6.4 MHz
Best	230 ms	71.9 ms
Worst	400 ms	125 ms

EXP – EXPONENTIAL



LN - Natural Logarithm

LN is a subroutine that takes the natural logarithm of a 32-bit binary floating-point number in registers (R(MQ) ; R(AC)). R(MQ) stores the exponent and the most significant 8 mantissa bits. R(AC) contains the remaining 16 mantissa bits. The result will be normalized and stored in registers (R(MQ) ; R(AC)).

1. **To Call LN:**

```
SEP CALL
,A(LN)
```

2. **Registers Used by LN:**

R(AC), R(MQ), R(MA), R(CN), R(SR), R(TP)

3. **Error Indication:**

By definition, the natural logarithm of zero and negative numbers is undefined. Thus, if the argument is either zero or negative, DF is set upon return to a calling program.

4. **Number Range:**

Domain X:

$7FFFFF \times 2^{80} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$0.293914 \times 10^{-38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y:

$A74701 \times 2^{07} \leqslant Y_{16} \leqslant 58078E \times 2^{07}$

$-88.7227 \leqslant Y_{10} \leqslant 88.0294$

5. **Other Subroutines Used:**

PHZ, PHST, PHSTR, POPR, FADD, FSUB, CORDIC

6. **Time:**

	2 MHz	6.4 MHz
Best	250 ms	78.1 ms
Worst	600 ms	188 ms

SQRT - Square Root

SQRT is a subroutine that takes the square root of a binary floating-point number stored in registers (R(MQ) ; R(AC)). The result is stored back into the same set of registers. The exponent and the most significant 8 mantissa bits are stored in R(MQ). R(AC) stores the least 16 mantissa bits.

1. **To Call SQRT:**

```
SEP CALL
,A(SQRT)
```

2. **Registers Used by SQRT:**

R(AC), R(MQ), R(MA), R(CN), R(SR), R(TP)

3. **Error Indication:**

By definition, the real root of a square-root function is defined only if the argument is non-negative. Thus, the DF flag is set if the binary floating point number in (R(MQ) ; R(AC)) is negative. Otherwise DF is reset to zero.

4. **Number Range:**

Domain X:

$0 \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$

$0 \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$

Range Y:

$0 \leqslant Y_{16} \leqslant 5A821B \times 2^{40}$

$0 \leqslant Y_{10} \leqslant 0.130436 \times 10^{20}$

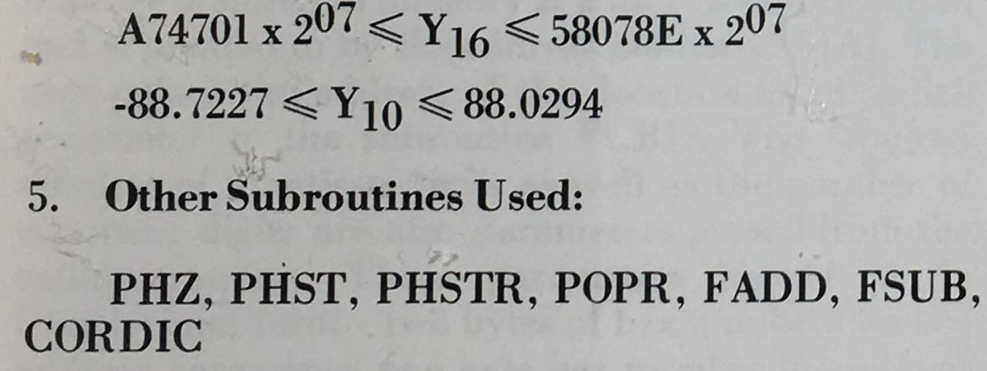
5. **Other Subroutines Used:**

LN, EXP

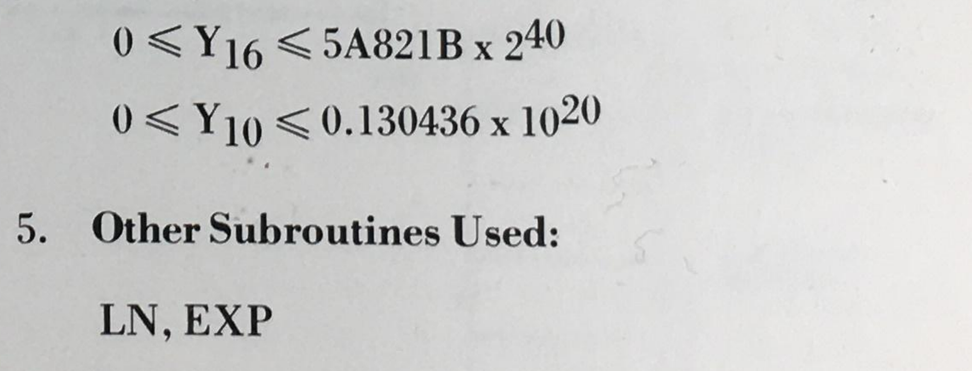
6. **Time:**

	2 MHz	6.4 MHz
Best	495 ms	155 ms
Worst	1 s	312 ms

LN – NATURAL LOGARITHM



SQRT – SQUARE ROOT



Detailed Description of Format-Conversion Subroutines

FCBD - Binary to Decimal

The subroutine FCBD is a format-conversion routine that converts a 32-bit binary floating-point number in registers (R(MQ) ; R(AC)) into an equivalent floating-point number in decimal and stores it one digit per byte in memory. See Fig. 11. The digit is stored in the lower 4 bits of the 8-bit byte. The exponent and the most significant 8 bits of the mantissa of the binary floating-point number come from R(MQ). The remaining 16 binary mantissa bits come from R(AC).

After the conversion, the decimal floating-point number is stored in memory at a user-defined location and is pointed to by the address pointer R(MA). The user passes the address of this location as an in-line parameter to the subroutine FCBD. The required number of mantissa digits as well as the number of exponent digits are also parameters passed from the calling routine. These parameters should be in hexadecimal form - two bytes of hex numbers for the address parameter; one byte hex number for each of the two digit-length parameters.

Two memory bytes, in addition to the total number of mantissa and exponent digits, are used to indicate the signs of mantissa and exponent of the decimal floating-point number. These two sign bytes are placed in front of the mantissa digits and exponent digits, respectively. A decimal point is assumed to the right of the mantissa sign byte.

If the number of actual converted mantissa digits is less than the digit-length parameter passed from the calling routine, zeros are padded on the right of the mantissa digits. In the similar case for the exponent, zeros are padded on the left of the exponent digits.

However, if the actual number of mantissa digits is greater than the digit-length parameter, then only the requested number of the most significant digits is converted and stored in memory. On the other hand, if the actual number of exponent digits is greater than the exponent digit-length parameter, the flag DF is

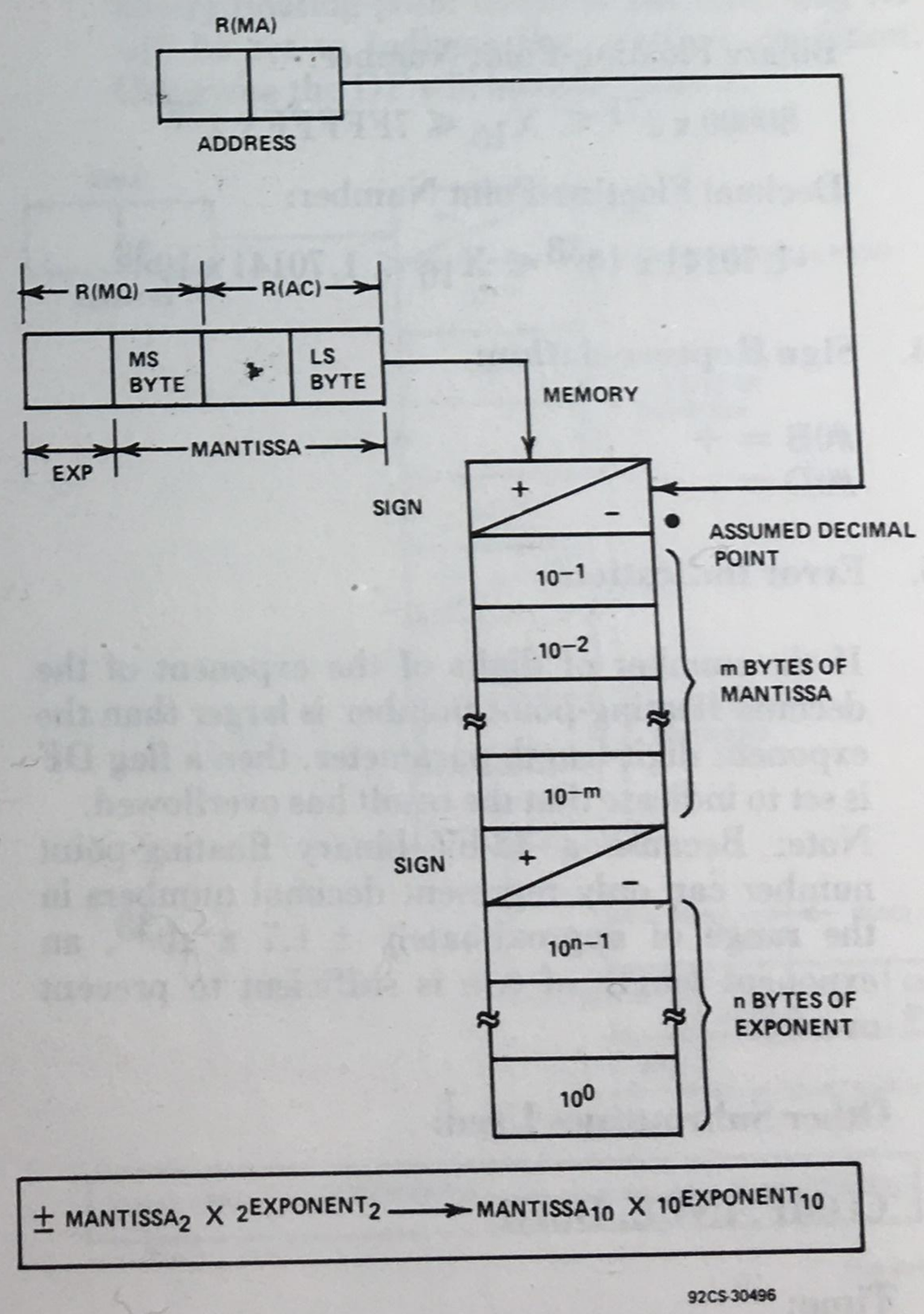


Fig. 11 – Operation of subroutine for converting a binary floating-point number to a decimal floating-point number.

set to indicate that either an overflow or an underflow condition has ocurred. The control is then returned to the calling program.

1. **To Call FCBD:**

```
SEP CALL
,A(FCBD)
,A(BUZZ)   ..Address where the decimal float-
           ..ing point number is to be stored.
,#m        ..One byte parameter. The number
           ..of mantissa digits required
           ..(excluding sign).
,#n        ..One byte parameter. The number
           ..of exponent digits required
           ..(excluding sign).
..After the conversion, R(MA) will point
..to the sign of the converted decimal floating-
..point number. See Fig. 11.
```

2. **Registers Used by FCBD:**

   R(AC), R(MQ), R(MA), R(CN), R(SR)

3. **Number Range:**

   Binary Floating-Point Number:

   $$80000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$$

   Decimal Floating-Point Number:

   $$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$$

4. **Sign Representation:**

   #0B = +
   #0D = –

5. **Error Indication:**

   If the number of digits of the exponent of the decimal floating-point number is larger than the exponent digit-length parameter, then a flag DF is set to indicate that the result has overflowed.
   Note: Because a 32-bit binary floating-point number can only represent decimal numbers in the range of approximately $\pm 1.7 \times 10^{38}$, an exponent length of two is sufficient to prevent overflow.

6. **Other Subroutines Used:**

   COMP, INCE, DSHL

7. **Time:**

	2 MHz	6.4 MHz
Best	7.5 ms	2.34 ms
Worst	150 ms	46.9 ms

Example 1: Convert the binary floating-point number 580000×2^{FF} [$= (1/2 + 1/8 + 1/16) \times 2^{-1} = 0.34375 \times 10^{00}$] into the decimal floating-point representation:

```
SEP CALL
,A(FCBD)   ..R(MQ) contains FF58.
           ..R(AC) contains 0000.
,A(BUFF)   ..The equivalent decimal
           ..floating-point number is
           ..to be stored at BUFF
           ..in memory.
,#04       ..4 digits of mantissa
           ..accuracy are required.
,#02       ..2 digits of exponent
           ..accuracy are required.
..After the conversion, the decimal floating-
..point number is stored in memory at
..address BUFF as shown in Fig. 12.
..Note - the last mantissa digit ("5") has been
..truncated because only 4 digits were requested.
```

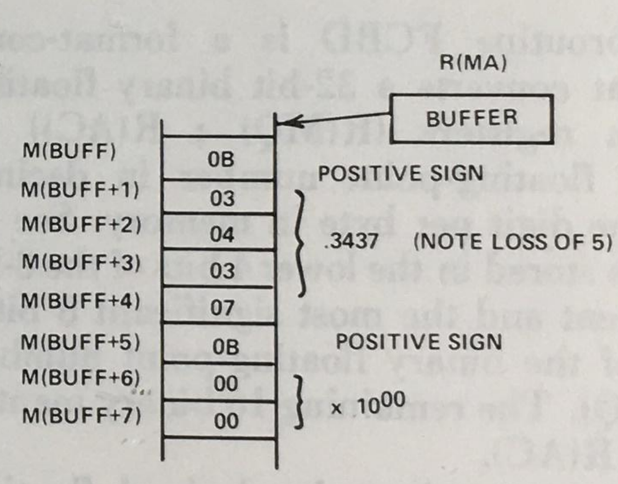


Fig. 12 – Memory after conversion of 580000×10^{00} to decimal floating-point representation (0.34375×10^{00}).

Example 2: Convert the number 740000×2^{09} [$= (1/2 + 1/4 + 1/8 + 1/32) \times 2^{9} = 0.464 \times 10^{03}$] to decimal floating-point representation:

```
SEP CALL
,A(FCBD)   ..Register R(MQ) contains 7400.
           ..Register R(AC) contains 0000.
,A(BUFF)   ..The decimal floating-point
           ..number is to be stored at
           ..BUFF in memory.
,#05       ..5 digits of mantissa are required
,#02       ..2 digits of exponent are required.
..After the conversion, the decimal floating-
..point number is stored in memory at
..address BUFF as shown in Fig. 13.
```

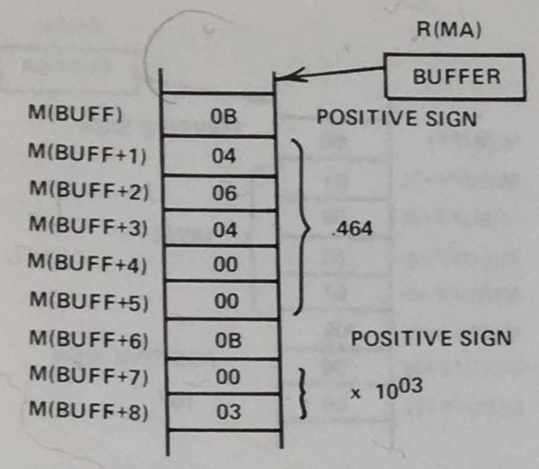


Fig. 13 — Memory after conversion of 740000 x 2^{09} to decimal floating-point representation (0.464 x 10^3). Note that zeros are padded on the left of the exponent digit, and on the right of the mantissa.

FCDB - Decimal to Binary

The FCDB format-conversion subroutine converts a decimal floating-point number in memory into an equivalent binary floating-point number and stores it in registers (R(MQ) ; R(AC)). The decimal floating-point number is stored in memory one byte per digit. See Fig. 14. A decimal point is assumed on the right of the sign byte.

The address of the floating-point number stored in memory is passed from the calling program to the FCDB subroutine as an in-line parameter. In addition to the address parameter, the calling program should provide the lengths of the mantissa and the exponent digits to the subroutine as two parameter bytes in hexadecimal form.

After the conversion, the binary floating-point number is stored in registers (R(MQ) ; R(AC)). R(MQ) contains the exponent and the most significant 8 mantissa bits. R(AC) contains the least significant 16 mantissa bits.

An error flag, DF, will be set if the decimal floating-point number is too large or too small to be represented as a 32-bit binary floating-point number. Otherwise the flag is reset to zero.

1. To Call FCDB:

```
SEP CALL
,A(FCDB)
,A(BUFF)   ..Address where the decimal
           ..floating-point number
           ..is to be stored.
,#m        ..One byte parameter. The length
           ..of the mantissa digits.
,#n        ..One byte parameter. The length
           ..of the exponent digits.
```

2. Registers Used by FCDB:

R(AC), R(MQ), R(MA), R(CN), R(SR), R(TP)

3. Number Range:

Decimal Floating-Point Number

$$-1.70141 \times 10^{38} \leqslant X_{10} \leqslant 1.70141 \times 10^{38}$$

Binary Floating-Point Number

$$80000 \times 2^{7F} \leqslant X_{16} \leqslant 7FFFFF \times 2^{7F}$$

4. Sign Representation:

0B = +
0D = −

5. Error Indication:

If the decimal floating-point number is either too large or too small to be represented as a 32-bit binary floating-point number, the error flag DF will be set to indicate the overflow condition. Otherwise the DF will be reset to zero.

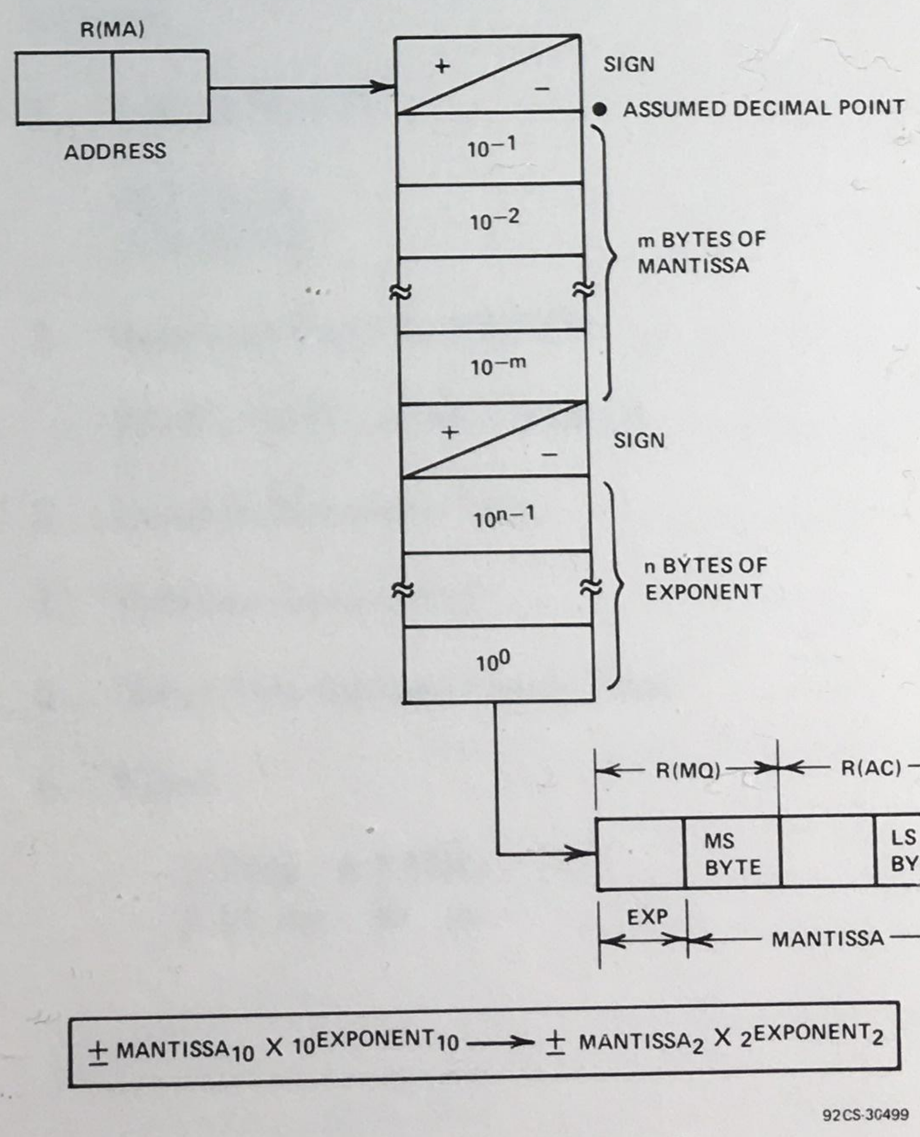


Fig. 14 — Operation of a subroutine for converting a decimal floating-point number to a binary floating-point number.

6. **Other Subroutines Used:**

DSHL, COMP, FADD, FMPY, FDIV

7. **Time:**

	2 MHz	6.4 MHz
Best	24 ms	7.5 ms
Worst	5 s	1.6 s

Example: Convert the decimal floating-point number 0.18375×10^0 into a binary floating-point representation. Fig. 15 shows memory before conversion.

```
SEP CALL
,A(FCDB)
,A(BUFF)   ..Address where the
           ..decimal point numbers are
           ..stored.
,#05       ..5 mantissa digits
,#01       ..1 exponent digit
```

The contents of registers R(MQ) and R(AC) after conversion of 0.18375×10^0 to binary are shown in Fig. 16. The binary floating-point number is 60000×2^{FE}.

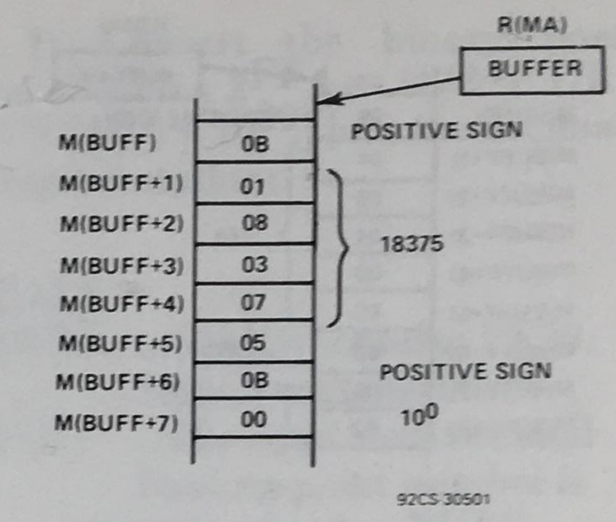


Fig. 15 – Memory before conversion of decimal floating-point number 0.18375×10^0.

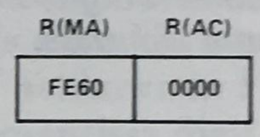


Fig. 16 – Registers R(MQ) and R(AC) after conversion of 0.18375×10^0 to binary.

Detailed Description of Utility Subroutines

COMP - Negation

This subroutine complements the binary floating-point operand in registers R(MQ) and R(AC). The result is placed back into the same registers.

1. **To Call COMP:**

```
SEP CALL
,A(COMP)
```

2. **Registers Used by COMP:**

   R(SR), R(AC), R(MQ)

3. **Error Indications:** None

4. **Number Range:** Full

5. **Other Subroutines Used:** None

6. **Time:**

2 MHz	6.4 MHz
0.2 ms	63 μs

PHSTR - Registers to Stack

Subroutine PHSTR pushes the contents of R(MQ) and R(AC) onto the stack, M(R(SP)). They are stored with R(MQ).1 at the top and decrease to R(AC).0. R(SP) is left pointing to a free RAM location.

1. **To Call PHSTR:**

```
SEP CALL
,A(PHSTR)
```

2. **Registers Used by PHSTR:**

   R(SP), R(SR), R(AC), R(MQ)

3. **Error Indications:** None

4. **Number Range:** Full

5. **Other Subroutines Used:** None

6. **Time:**

2 MHz	6.4 MHz
0.14 ms	45 μs

POPR - Memory to Registers

Subroutine POPR loads the four bytes pointed to by R(MA) and places them into R(MQ).1, R(MQ).0, R(AC).1 and R(AC).0 in sequence. R(MA) must point to R(MQ).1 before call. Upon return from the call, R(MA) is left pointing to the byte beyond R(AC).0.

1. **To Call POPR:**

```
SEP CALL
,A(POPR)
```

2. **Registers Used by POPR:**

   R(MA), R(MQ), R(AC)

3. **Error Indications:** None

4. **Number Range:** Full

5. **Other Subroutines Used:** None

6. **Time:**

2 MHz	6.4 MHz
72 μs	22.5 μs

PHZ - Zero to Stack

Subroutine PHZ pushes four zero bytes onto the stack, M(R(SP)). On return, R(MA) is left pointing to the topmost zero byte.

1. **To Call PHZ:**

```
SEP CALL
,A(PHZ)
```

2. **Registers Used by PHZ:**

   R(SP), R(SR), R(MA)

3. **Error Indications:** None

4. **Number Range:** Full

5. **Other Subroutines Used:** None

6. **Time:**

2 MHz	6.4 MHz
0.17 ms	52.5 μs

PHST - Constants to Stack

Subroutine PHST pushes a four-byte constant onto the stack M(R(SP)), with the most significant byte pushed on last. On return, R(MA) is left pointing to the most significant byte.

1. **To Call PHST:**

```
SEP CALL
,A(PHST)
,#LSB, XX, XX, MSB
```

2. **Registers Used by PHST:**

   R(SP), R(SR), R(LINK) R(MA)

3. **Error Indications:** None

4. **Number Range:** Full

5. **Other Subroutines Used:** None

6. **Time:**

2 MHz	6.4 MHz
0.18 ms	55 μs

DECMA - Decrease R(MA)

Subroutine DECMA decreases the value of R(MA) by an amount passed as an in-line parameter from the calling routine.

1. **To Call DECMA:**

```
SEP CALL
,A(DECMA)
,#XX   ..Number of times to decrement
       ..R(MA) in hex.
```

2. **Registers Used by DECMA:**

   R(LINK), R(MA)

3. **Error Indications:** None

4. **Number Range:** Full

5. **Other Subroutines Used:** None

6. **Time:**

2 MHz	6.4 MHz
80 μs	25 μs

Accuracy and Errors

System Errors

In binary floating-point arithmetic, the numbers are seldom represented exactly by a given number of binary bits because of the nature of binary floating-point representation. For instance, the number 0.5 in decimal is represented exactly by 0.1 in binary. However, the number 0.3 in decimal cannot be exactly represented in binary by the 24 bits of the mantissa. Instead 0.3_{10} is stored as

$$0.100\ 1100\ 1100\ 1100\ 1100\ 1100 \times 2^{1111\ 1111}$$

which is 0.2999 9999 98 in decimal. The error is $0.2_{10} \times 10^{-9}$.

It is important to realize that in the two number systems, fixed-point and floating point, the number fields are different. In the floating-point system the numbers are bunched around zero. In the fixed-point system the numbers are uniformly distributed over the range of the number field.

To illustrate the difference consider a register, for simplicity, only 8 digits long. If there is a fixed decimal point in the third place, for example, the smallest increment that can be added to any digit is 0.001_{10}. The fixed-point numbers range from 0.001_{10} to 99999.99_{10}. The difference between numbers no matter where a number is in the range, is 0.001. Thus the numbers are uniformly distributed over the range of fixed-point numbers.

Next, consider that the decimal point "floats". In the eight-digit example, the smallest number that can be added to zero in floating-point notation is 0.00000001_{10}. Consider incrementing a floating-point number when the decimal point is after the far-right digit. The smallest number that can be added to 99999998 is 1. The difference in the size of the smallest number when incrementing a full and empty register in floating-point notation is a factor of 10^8. In the floating-point system the range of numbers is from 99999999 to 0.00000001. When the numbers are very small the distance between them is 0.0000 0001, and when the register is full the difference between the numbers is 1. Clearly, over the range of floating-point numbers, the distribution is not uniform; actually there are as many numbers grouped between 0 and 1 as there are between 1 and the full register size.

During floating-point numerical computation, other types of errors can also be introduced. For example, when two numbers with unequal exponents are added or subtracted, the mantissa bits of the smaller operand must be shifted right until the exponents are identical. The shifted-out mantissa bits are lost during the alignment process.

These types of errors are usually inherent in the floating-point arithmetic representation or in the process of computation. The user should be aware that such errors are unavoidable and are inherent in floating-point arithmetic.

User-Caused Errors

The user can also cause error conditions by passing an argument to a subroutine so that the arithmetic function becomes undefined. Examples would be an attempt to take the square root of a negative number, the natural logarithm of zero, and the like.

The subroutines will set the data flag DF to 1 if the error conditions are caused by the user. The DF flag will also be set if overflow or underflow occurs, which is possible during addition, subtraction, multiplication, division, exponential, and format-conversion routines.

Accuracy

In binary floating-point arithmetic, if the error in mantissa bits is less than 2^{-n}, the binary number is said to have n-bit accuracy provided that its exponent is exactly correct. For example, if the computed result is 0.110011×2^{011} and the desired result should have been 0.110100×2^{011}, the error from the computation is 0.000001×2^{011}. Thus, the result has 5-bit accuracy because the difference in the mantissa bits is 2^{-6}.

In decimal floating-point arithmetic, the number is said to have n-digits of accuracy, if the exponent is correct and if the error in the mantissa is less than $\pm 10^{-n}/2$. For example, if the computed result is 0.1238 and the correct result is 0.1235, the difference is 0.0003. Because the error is less than $10^{-4}/2$, three-digit accuracy was obtained.

Note, however, that if the computed result is very small, the n-bit or n-digit accuracy, as previously defined, does not necessarily give the best indication of computing accuracy.

Performance Data

The following computations were used to test the accuracy of various functions included in this software package.

$\sin X^2 + \cos^2 X = 1, X = 0 \rightarrow 700; \quad \Delta X = 0.01$

$X = \mathrm{Ln}(e^X), X = -88 \rightarrow +88; \quad \Delta X = 0.01$

$X = \arctan\left(\frac{\sin X}{\cos X}\right), X = 0 \rightarrow 90; \quad \Delta X = 0.01$

$X = \sqrt{X^2}, X = 0 \rightarrow 4700; \quad \Delta X = 0.01$

Arguments (for the range and increments shown above) in binary floating-point form were passed to individual functions, and the results were compared with the original argument values (or 1 in the $\sin^2 X + \cos^2 X$ case). It was observed that a 17-bit (or better) accuracy was obtained from the above computations. A 17-bit accuracy in binary floating-point arithmetic corresponds to approximately 5-digit decimal floating-point arithemtic.

Demonstration Program

A demonstration program that simulates a calculator is included on the diskette CDP18S827 in both level I assembly source and object code. The numbers for the calculator program may be entered in free format with up to eight mantissa and two exponent digits. The program is loaded and executed in the following manner:

1. Load the program using the "$L" command for the CDS II (CDP18S005) or $P8600 for the CDS (CDP18S004).
2. Type "$U1" for the CDSII or $P1 for the CDS to start the program after it has been loaded. The program then asks if the user wishes further instructions. If the user types a "Y" the instructions are printed.
3. After the instructions have been printed or, if the response to 2 above was an "N", the program issues the user prompt "?". The first operand may now be entered. A positive sign is assumed for both the mantissa and the exponent if no sign is entered. The up arrow "↑" (or ^) is used to delimit the exponent. Additionally, all spaces are ignored.
4. Once the operand has been entered, the appropriate symbol for the desired function, as indicated in the instructions, is typed. At this point, the operand is converted into floating-point binary. If only one operand is required, an = sign is printed followed by the results in both decimal and its floating-point hexadecimal equivalent. If a second operand is required, the system waits for further input. The second operand must be terminated with an = sign. The function is then performed, and the results printed as above.
5. After printing the results, the system re-issues the user prompt. The previous result is retained, allowing chain calculations.

Two additional features of the Demonstration Program allow the user to obtain the floating-point hexadecimal equivalent of a decimal number and the decimal equivalent of a floating-point hexadecimal number. These operations are invoked by entering the decimal number followed by an = sign to obtain the hexadecimal equivalent or by entering a # sign followed by eight hexadecimal digits to obtain the decimal equivalent.

References

CORDIC

J. E. Volder, "The Cordic Trigonometric Computing Technique," **IRE Transactions On Electronic Computers**, Vol. EC-8, No. 3, pp 330-334, Sept. 1959.

J. S. Walther, "A Unified Algorithm for Elementary Functions," **IEEE Spring Joint Computer Conference**, 1971, pp 379-385.

NUMBER SYSTEMS

C. L. Conrad, **COMPUTER MATHEMATICS**, Hayden Book Co. Inc., 1976.